SpartanMC

Memory Organization

Table of Contents

List of Figures

List of Tables

Memory Organization

The SpartanMC main memory is a compound of single memory blocks of 2k rows with 18 bit width. The number of blocks and therefore the size of the main memory is configurable. The memory blocks are implemented by using the FPGA internal BlockRAMs. Each block consists of two FPGA BlockRAMs of 2k rows and 9 bit width. Since the FPGA BlockRAMs are dual ported, one port is used to read instructions and the other port is used to read and write data.

The SpartanMC stores data in big endian byte order.

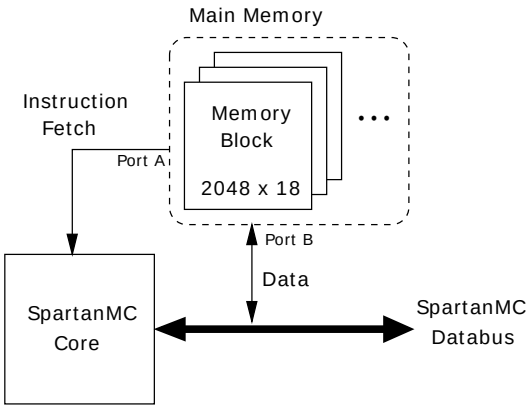

Figure 1: Dual ported main memory

1. Address Management

Each port of the main memory is connected to a 18 bit address bus. Since the main memory consists of 2k x 18 bit blocks, there are 11 bit required to address the rows within a block. The remaining 7 bit of the address bus are used to select the memory block. Therefore a possible maximum of addressable memory of 256k of 18 bit words distributed to 128 memory blocks could be instantiated. For the instruction port of the memory, the program counter (PC) is used as address bus.

For better memory utilization of the data section the data port provides a 9 bit wise memory access. Therefore the least significant bit (Align) of the data address bus is used to select the upper or lower half word which is used in load and store instructions (l9,s9). The remaining 17 bit are used to address the lower 128k of the memory. To address the upper 128k, the content of SFR_STATUS_7(MM) is used as most significant bit of the data address bus.

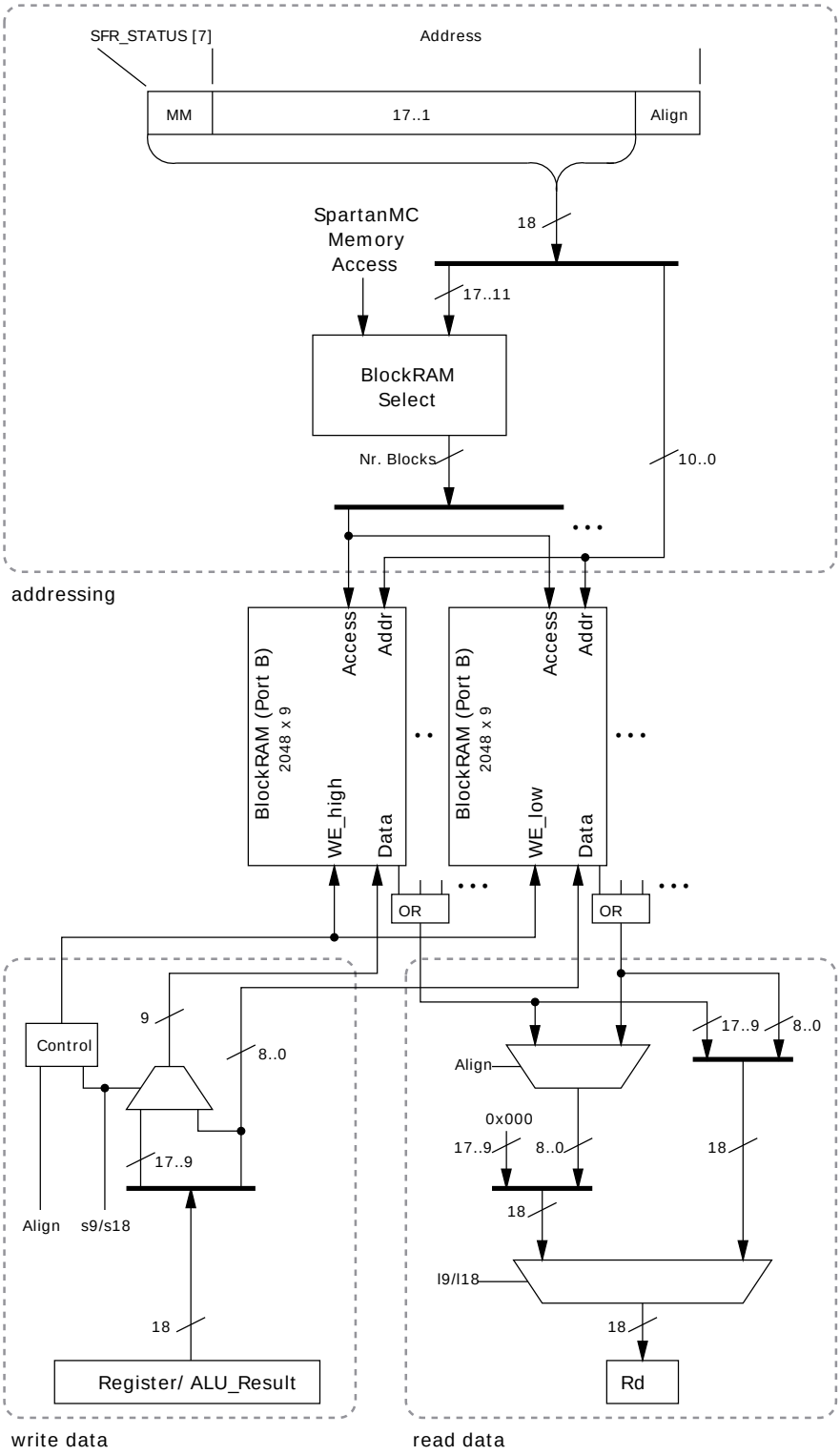

Figure 2: Data address management

Note: Due to the 9 bit wise data access, the correct address assignment of data addresses in assembler code has to be assured. The address value of the data address has to be twice the size of the regular instruction address.

2. Peripheral Access

2.1. Memory Mapped

Peripherals are connected to the regular data and address bus of the SpartanMC. Thus, peripheral devices are mapped to the SpartanMC address space at a dedicated address (IO_BASE_ADR). For exchanging small amounts of data between processor and peripheral, peripherals can provide a set of 18 bit registers. These registers are implemented as distributed memory on the FPGA.

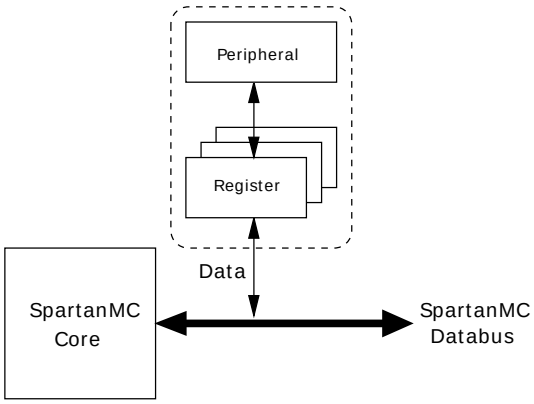

Figure 3: Memory mapped registers

The upper 8 bit part of the 18 bit address is used to select the peripheral address space. The selection is carried out by comparing the upper 8 bit part of the current address with the upper 8 bit of the configured base address (IO_BASE_ADR). The lower 10 bit are used to select the peripheral register within this address space. Therefore the 10 bit are divided into two parts: the first 9..n bit to access the correct peripheral module according to the BASE_ADR of the module and the second n-1..0 bit to access the 18 bit register within this peripheral module. The value of n depends on the number of registers provided by the peripheral (e.g. a value of n=3 implies a maximum of 8 registers for that module).

Note: The base address of the peripheral modules should be sorted by the number of registers. Starting with the peripheral using the most registers. This scheme avoids the overlapping of address spaces between different peripherals.

The data access to the registers is similar to the access to the main memory. For reading data (l9/l18) the align bit (LSB of the address) can be used to select the upper or lower half word of the register. For writing data the align bit is meaningless therefore only the s18 operation can be performed on peripheral memory.

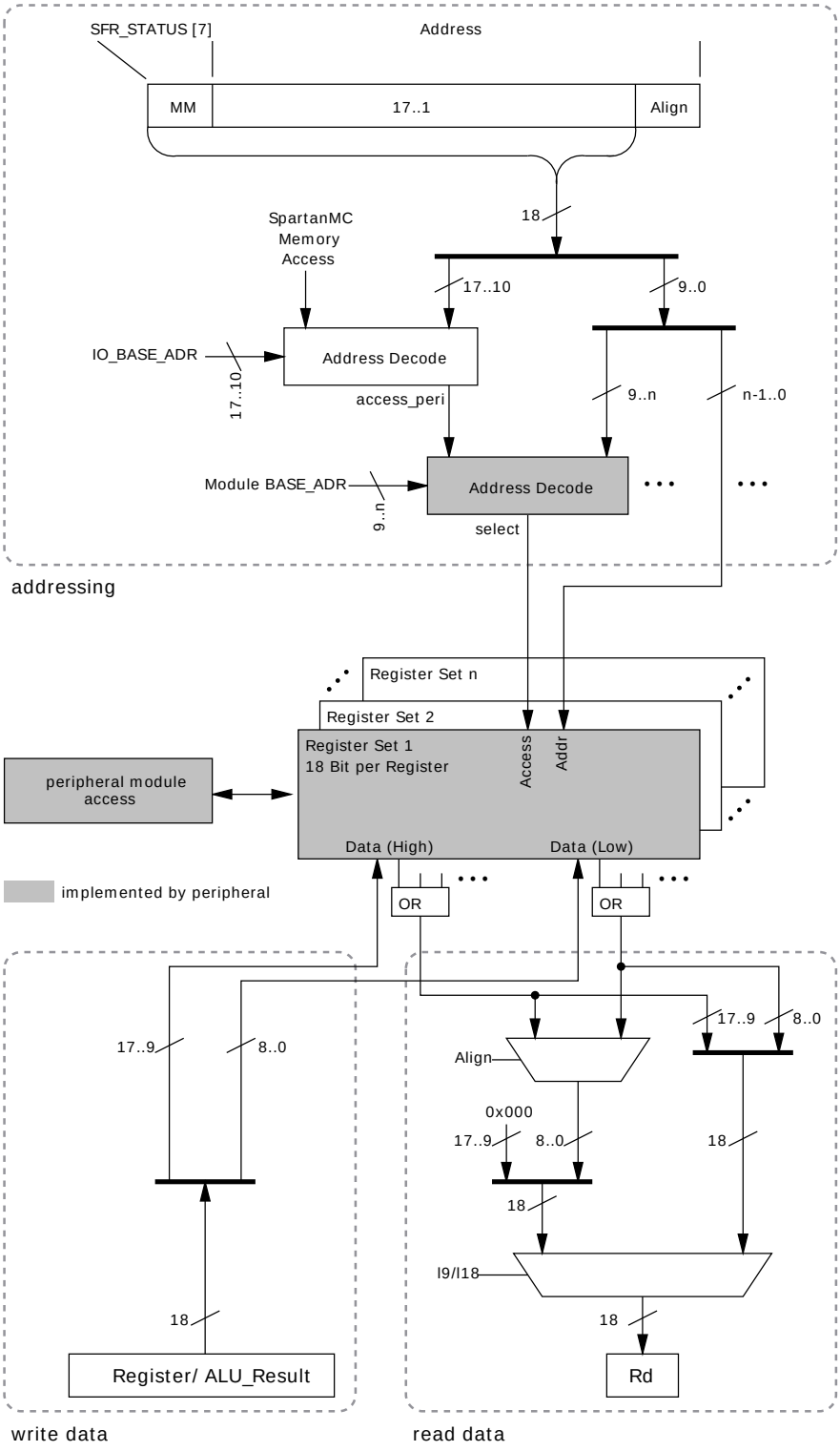

Figure 4: Peripheral register address management

2.2. Direct Memory Access (DMA)

Peripherals that work on large volumes of data can use BlockRAMs as data interface to the processor. In this case the first port is connected to the SpartanMC address and data bus and the second port is connected to the peripheral which works autonomously on the data in the memory block. This can be regarded as DMA style operation.

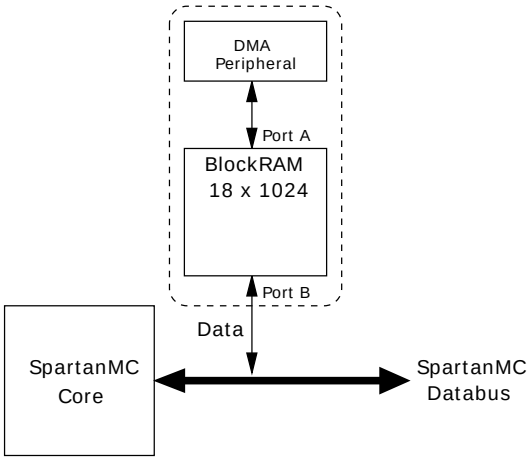

Figure 5: DMA with dual ported BlockRAM

Note: Due to the SpartanMC memory management which uses the second port of the BlockRAM as instruction fetch, the processor can not execute code from the DMA memory since the second port is used as peripheral interface. This missing master mode DMA would lead to copying overhead if data needs to be buffered between processing it with different peripherals.

The upper 8 bit part of the 18 bit address is used to select the DMA device. The selection is carried out by comparing the upper 8 bit part of the current address with the upper 8 bit of the configured base address (DMA_BASE_ADR). The lower 10 bit are used to select the row within the DMA BlockRAM.

The data access to the DMA memory is similar to the access to the main memory. For reading data (l9/l18) the align bit (LSB of the address) can be used to select the upper or lower half word of the register. For writing data, the half word to be written is selected by the store_access_low and store_access_high lines.

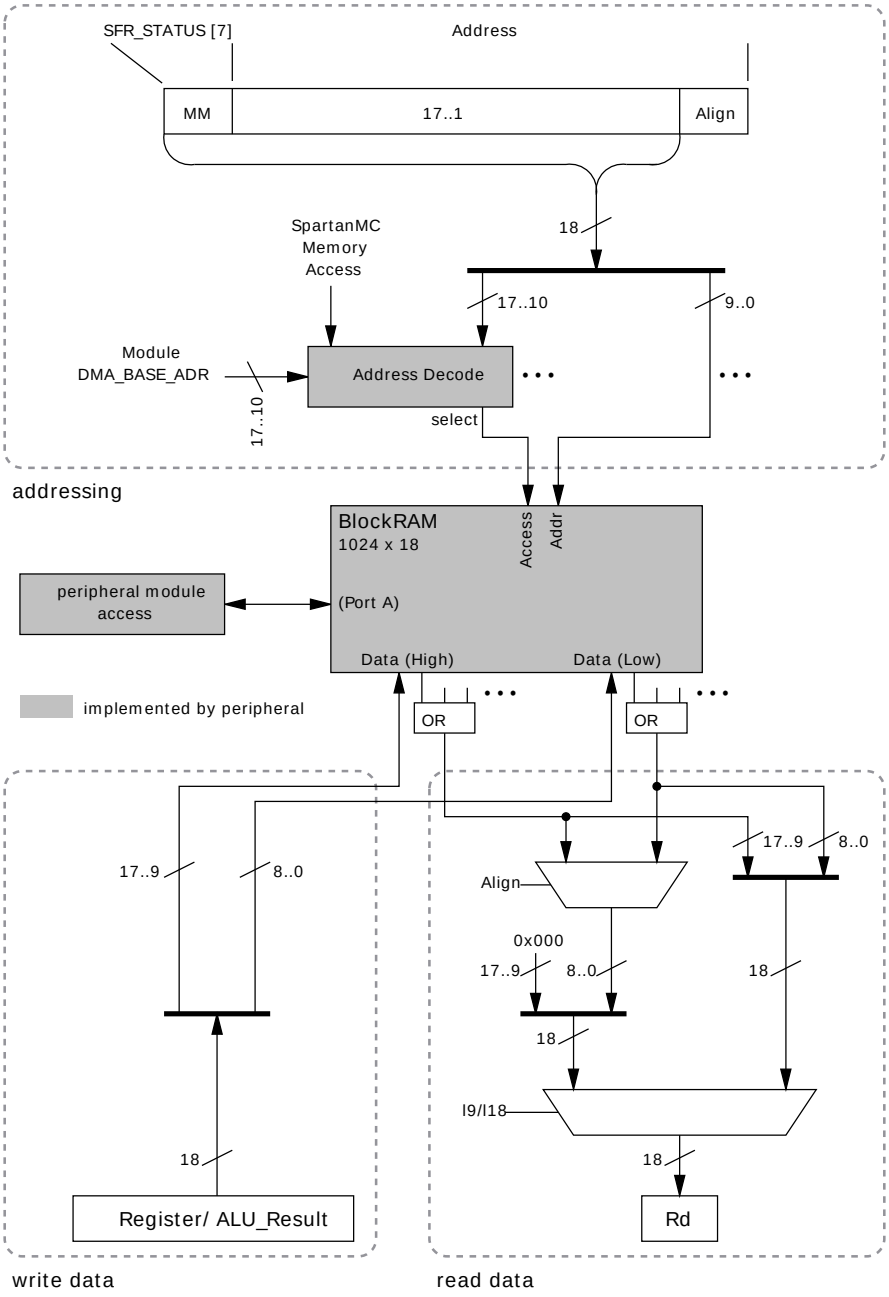

Figure 6: DMA address management

2.3. Data Read Interface

The main memory blocks and the peripheral memory are connected to the data memory interface of the processor core. In order to avoid tri-state buffers, all incoming data is combined through a wide or-gate. Thus, all memory blocks and peripherals that are currently not addressed must provide a value of zero on their outputs.

3. Data and Code Buses

This section describes the various signals making up the code and data buses and their timing.

3.1. Data Bus

Table 1: Data Bus Signals

Signal	Source Element	Descripton
clk	Core	Clock
reset	Core	Reset Signal
mem_access	Core	Signals that the address is valid and a read / write should be performed
mem_wr_low / mem_wr_high	Core	Write the lower / upper halfword to memory. Only relevant if mem_access is set.
mem_addr_high / mem_addr_block	Core	Memory address in bytes. Only relevant if mem_access is set.
mem_di	Core	Data to write. Only relevant if mem_access and mem_wr_* are set.
mem_busy	Memory	The memory cannot answer a pending request yet. Processor needs to stall.
mem_do	Memory	The data read from memory.

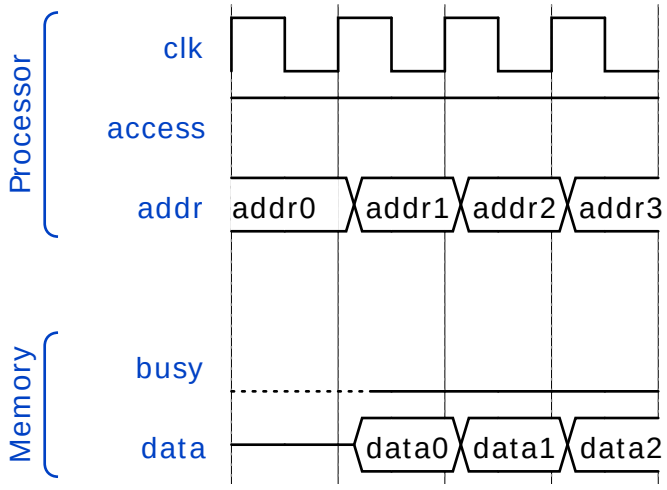

Figure 7: Data Bus Access without mem_busy

As shown above, read data needs to be output in the cycle following the request. Data always is read from memory, even if one or both of the write signals are set. The old data is expected to be read from the memory. This is needed for the swap instruction.

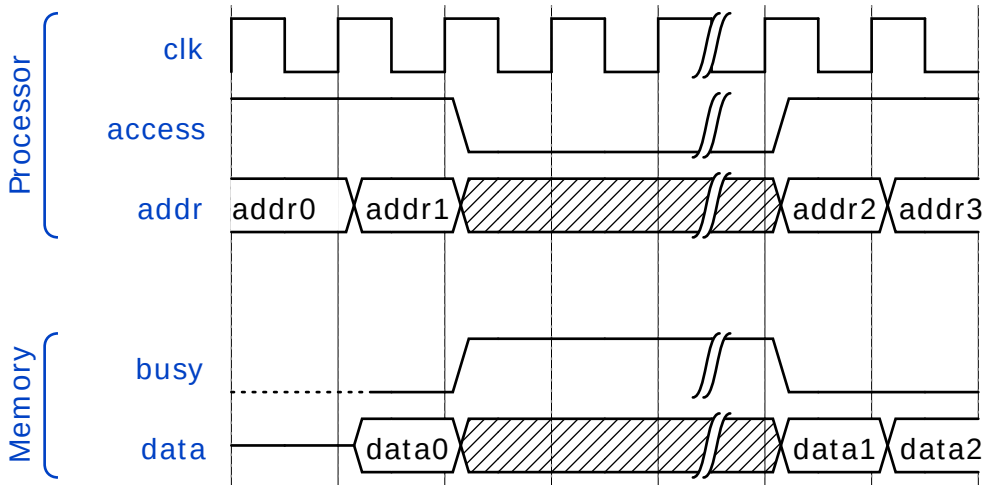

Figure 8: Data Bus Access with mem_busy

If a request cannot be served directly, mem_busy needs to be asserted until the request is finished. mem_do needs to be driven in the cycle where mem_busy is deasserted. mem_access is guaranteed to not be asserted until mem_busy is deasserted. Therefore, a combinatoric path from mem_busy to mem_access exists inside the processor core. Memory modules therefore must take care not to introduce a combinatoric path from mem_access to mem_busy.

3.2. Code Bus

Table 2: Data Bus Signals

Signal	Source Element	Descripton
clk	Core	Clock
reset	Core	Reset Signal
code_access	Core	Signals that the address is valid and a read / write should be performed
code_addr	Core	Memory address in words. Only relevant if mem_access is set.
code_jmp	Core	The processor is performing a jump. TODO: When is this signal generated? Before or during jump?
code_busy	Memory	The memory cannot answer a pending request yet. Processor needs to stall.

Signal	Source Element	Descripton
code_di	Memory	The code read from memory.

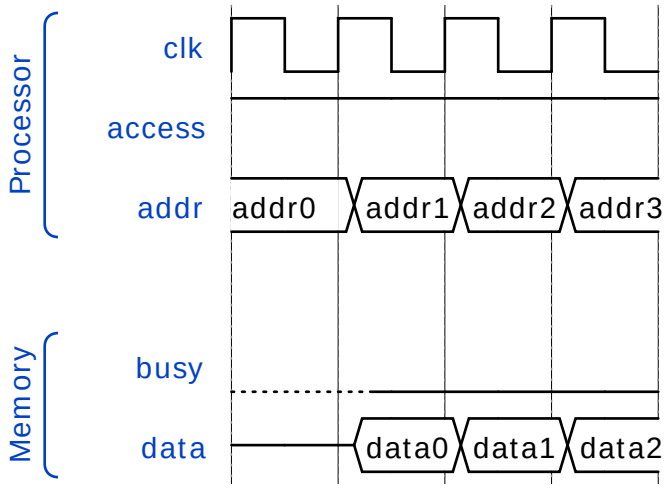

Figure 9: Code Bus Access without mem_busy

As shown above, read data needs to be output in the cycle following the request.

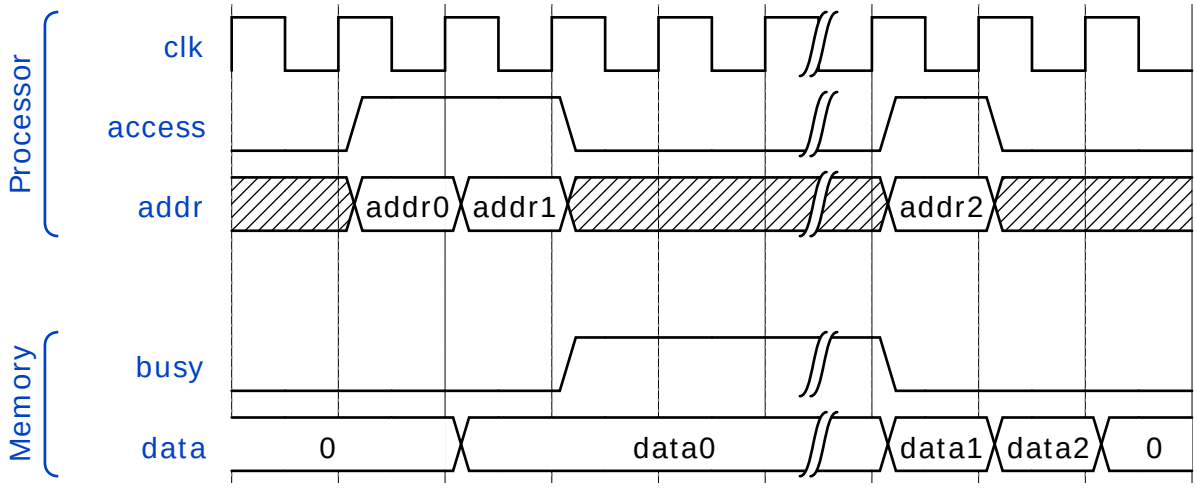

Figure 10: Code Bus Access with mem_busy

If a request cannot be served directly, mem_busy needs to be asserted until the request is finished. code_di needs to be driven in the cycle where mem_busy is deasserted. code_access is guaranteed to not be asserted until code_busy is deasserted. Therefore, a combinatoric path from code_busy to code_access exists inside the processor core. Memory modules therefore must take care not to introduce a combinatoric path from code_access to code_busy.

While code_busy is asserted, code_di needs to keep its previous value. For memories based on block RAMS, this can be achieved by driving its enable input to low.

4. Example Memory Map

The following image describes a memory map for an SpartanMC example system and an application using traps and interrupts. The specfic addresses of the different application parts (ISR, Traps, IRQ Handler etc.) are automatically defined through compiler tools. The start addresses of DMA memory (in this example 0x19000) can be defined in the hardware configuraion generated through jConfig.

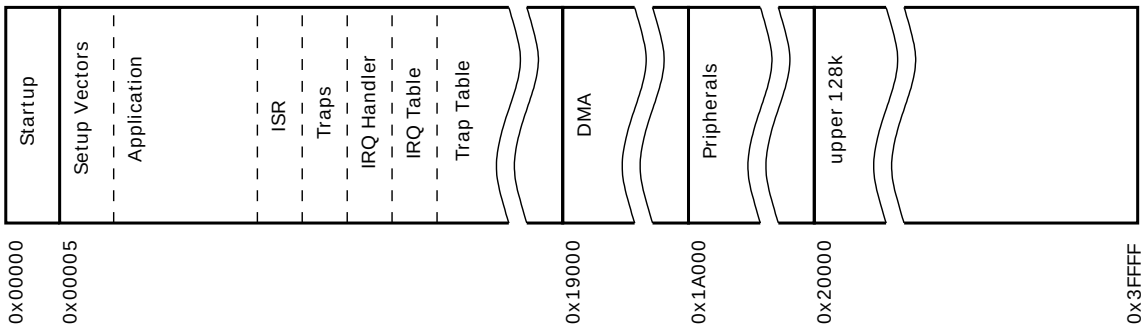

Figure 11: Example memory map

Startup: The startup code is generated by compiler tools at address 0x0000. It contains a branch to the application specific *Setup Vectors* -subroutine. The required branch address is defined within the system headers generated via jConfig.

Setup Vectors: Setup the address for the interrupt handler and the trap base address for this application.

Application: Contains the application code.

ISR: The interrupt service routines for the defined interrupts

Traps: The trap code for the defined traps.

IRQ Handler: Performs the IRQ prolog and epilog and links the IRQ table of the application.

IRQ Table: The interrupt branch table. Each 18 bit address contains the jump instructions to the interrupt code. The table length depends on the number of configured interrupts.

Trap Table: The branch table for traps. Each 18 bit address contains the jump instructions to a specific trap code. Since the the upper 10 bit are used as trap

base address a maximum of 255 traps can be defined using the lower 8 bit. The implemented table length depends on the number of traps defined in the application

DMA: The memory section for DMA capable peripherals. This memory section is 18 bit aligned and contains data only.

Peripherals: The memory section for memory mapped peripherals. The start address of this section has to be set beyond the actual configured main memory section.